Expertise
and insight
for the future

Bao Tran

Development of an Internet of Things platform for Smart Agricultures solutions

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

30 April 2020

Author Title Number of Pages Date	Bao Tran Development of an Internet of Things platform for Smart Agriculture solutions. 34 pages 30 April 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department (ICT)

Internet of Things recently receives a lot of attention as embedded devices and cloud technologies keep evolving rapidly. One of the most common IoT's application is in smart agriculture system, where cutting-edge technology is used to increase crop productivity and plant quality.

GFarming is a young startup company that focus on developing smart agriculture solutions. The purpose of this final year thesis was to prototype a platform for the company to integrate with existed sensors and gateways using Spring, actor models and Apache Kafka as core messaging system.

Another aim is to study about reactive system implementation using the actor model, its interaction patterns and Kafka as a distributed messaging system. Even though it is a prototype, the application can still be scaled and developed further easily with acceptable performance for large amount of end user's devices.

In conclusion, by applying reactive system concept with actor models, Kafka distributed messaging system and Cassandra as storage system, a fully functional and scalable prototype is enabled.

Keywords	Akka, Spring, IoT, Apache Kafka, Apache Cassandra

Contents

List of Abbreviations

1 Introduction 1

2 Background and Requirements 1

2.1 Internet of Things and Smart Agriculture solutions 1
2.2 Application Requirements 3

3 Design and Technology used 3

3.1 Reactive system and actor model 3
3.1.1 Akka 5
3.1.2 Event Sourcing and Akka Persistence 6
3.2 Kotlin 6
3.3 Apache Kafka 7
3.4 Apache Cassandra 9
3.5 Spring 11
3.5.1 Spring Framework 12
3.5.2 Spring Boot 14
3.5.3 Spring Data 14
3.5.4 Spring for Apache Kafka 14

4 Implementation 15

4.1 Project setup 15
4.2 Domain and Repository Layer 18
4.3 IoT Actor System 19
4.3.1 Application actor hierarchy 19
4.3.2 Sensor Persistence Actor 22
4.3.3 Kafka consumer’s implementation 24
4.3.4 Adding telemetry to sensor actor 25
4.3.5 Querying gateway sensors state 27
4.4 WebSocket Integration 28
4.5 Unit Testing 29

5 Testing and Conclusion 31

5.1 Testing 31
5.2 Conclusion 34
References 35

List of Abbreviations

IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
Wi-Fi	Wireless Fidelity
POJO	Plain old Java object
CI/CD	Continuous Integration and Continuous Delivery
JVM	Java Virtual Machine
IDE	Integrated Development Environment
STM	Software Transactional Memory
J2EE	Java 2 Platform, Enterprise Edition
API	Application Programming Interface
IOC	Inversion of Control
AOP	Aspected-oriented Programming
XML	Extensible Markup Language
MVC	Model, View, Controller

URL	Uniform Resource Locator

2 Introduction

Internet of Things nowadays is among the hot trend in technology. However, dealing with big amount of data generated from enormous embedded devices and sensors requiring a careful design and best practices.

This thesis shows a concrete implementation of an IoT platform that follow the concept of reactive system architecture using actor models and various modern application development technologies.

GFarming is a startup company that focuses on IoT platform with smart agriculture devices. The company's sensors will help farmers to efficiently control and automate their product's environment. Furthermore, by utilizing the sensors data, GFarming can provide the guidance for tree planting best practices or stimulate an artificial environment for many purposes.

The final year thesis is divided into five chapters. Firstly, chapter 1 provide an overall introduction to the thesis topic. Secondly, chapter 2 contains the backgrounds and functional requirements of the application. The following chapters describe the design and technology used and main features' implementation details of the thesis project including support for unit testing. Finally, the thesis is closed by some discussion related to the topic and a conclusion.

3 Background and Requirements

3.1 Internet of Things and Smart Agriculture solutions

The Internet of Things, or IoT, refers to the billions of physical devices around the world that are now connected to the internet, all collecting and sharing data. A complete IoT architecture usually is a combination of the same four components: devices, connectivity, platform, and an application. Things in Internet of Things are the sources for data coming from measurement or interaction with real world environment. IoT devices commonly should be designed for specialize tasks with cost and power consumption

considerations. In addition to an optional operating system, device allows an access to hardware through an abstract software layer. This abstraction simplifies the development process for device functionalities [1].

Furthermore, an important aspect of Things is the ability to support one or more communication with other Things or with the IoT gateway. The common communication protocols in IoT are serial, Bluetooth, MQTT, Wireless, etc. Finally, some devices may also support the remote-control functionality which lead user or manufacture upgrade, downgrade the firmware or checking battery level [1].

In addition, some devices can't connect directly to the central platform and require the use of an IoT gateway to bridge the gap between local environment and the cloud platform. The devices are usually linked to the gateway. The gateway then reads the necessary information and then transmit device's data to the IoT platform using cellular or Wi-Fi network connection. Additionally, gateways allow developers to integrate Edge Compute into the whole architecture. Edge Compute moves authority and processing from the Cloud closer to devices. The Cloud is a crucial part of the architecture, but it has certain restrictions, such as internet connection trustworthy or network latency [1].

Next, IoT Platform is the core IoT solution's data warehouse and orchestration engine. The IoT Platform may include many modules, including Data Ingestion Services, Data Warehousing, Workflows or Rules Engines, Dashboards, etc. [1].

Finally, the IoT solution offers the end-user experience of how end users communicate with the data gathered from the IoT devices. Available options can be a mobile or desktop application, a web interface, or a passive experience with no active communication. It can require a lot of effort to create those solutions. However, it can bring a lot of benefits as it is where the real value of the IoT solution is observed [1].

Smart agriculture systems clearly stand out from all IoT-enabled solutions. As one of the main sectors of the global economy, agriculture also boasts the most competitive level of IoT adoption. Agritech is a booming industry and, as of today, a wide range of smart farming systems allow farmers to meet their daily challenge. Planting, irrigation, seed selection and pest control – agricultural field monitoring gathers a range of indicators so farmers can take appropriate action to handle these tasks. Examples of smart agricultural

monitoring solutions include soil condition tracking, weather tracking, Greenhouse automation system, crop monitoring systems [2].

3.2 Application Requirements

The main function of IoT platform is consuming data from Kafka and provide some required features for end customer. In details, those features are:

- Support sensor types currently are air temperature, soil temperature and soil moisture. Concrete sensor unit and unit conversion are excluded and can be added in later phase.
- User can create gateway through rest endpoint.
- User can create one or multiple sensor with supported senor types for specific gateway.
- Telemetry data will be published to Kafka under telemetry topic; those data should be consumed and persisted as state.
- Latest telemetry for each sensor should be available as snapshot for web socket subscription or through rest endpoint.

In addition to the main functional requirements of the application, the design and implementation should keep in mind that the platform can be easily extend and scale according to load in the future.

4 Design and Technology used

4.1 Reactive system and actor model

The criteria for applications have changed significantly in recent years. Users expect a quick response time and a large volume of data is always coming. Therefore, a consistent approach to system architecture is required, and all the necessary aspects are described in the reactive manifesto. The Reactive Manifesto was released on 16 September 2014, stating that the Responsive, Resilient, Elastic and Message Drive system is the Reactive Process. In particular, reactive system is defined to be more versatile, loosely coupled and scalable. They can deal with failure, and approach it with grace

rather than catastrophe when failure occurs. Furthermore, reactive systems are responsive, meaning that they can provide users with effective interactive feedback. [3]

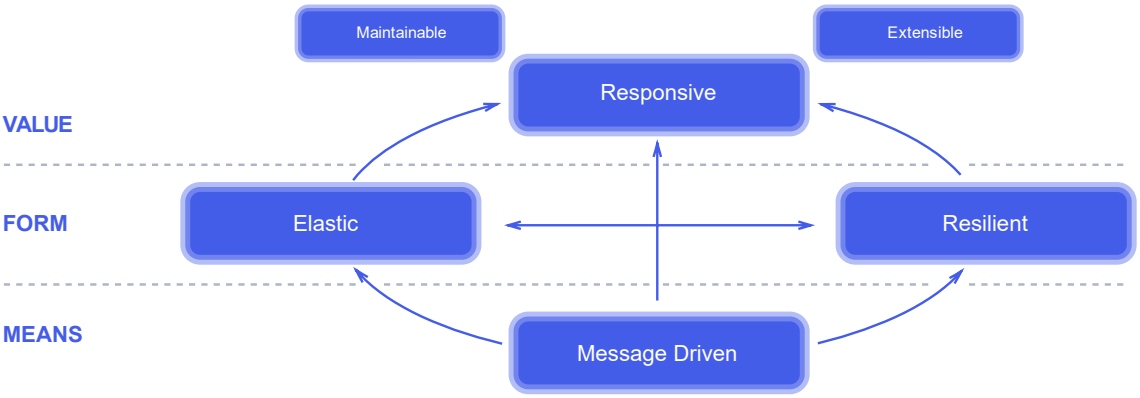


Figure 1. The structure of reactive values defined in reactive manifesto. Copied from [3].

As illustrated in Figure 1, the Reactive Manifesto defines the four main principles: Message Driven, Resilient, Elastic and Responsive. Firstly, message-driven is the foundational element of the Reactive System. This is the fundamental concept or procedure for the rest of the three principles. Specifically, in a reactive system, all components interact with each other by transmitting asynchronous messages. Secondly, elasticity means that the system can be scaled horizontally or vertically, depending on the application needs. Thirdly, by means of resilience, the reactive system will respond to the event of failure of users by recovering itself. Lastly, the main objective of the reactive system is responsiveness, which means that system should respond to the user in a timely manner. By applying the three characteristics of the reactive system: message-driven, elasticity, resilience, the main objective-responsiveness will be achieved. [3]

One of the key technologies that enables us to build microservice architectures that develop rapidly, can scale and run without stopping, is Actor model-based systems. It is the Actor model that provides the core functionality of the Reactive Systems as described in figure 1 [4, p.43]. Created in 1973 by renowned computer scientist Carl Hewitt, the Actor model defines a fairly simple yet efficient way to design and execute applications that can spread and share work across all device resources — from threads and cores to server clusters and data centers. The Actor Model solve the problem with concurrency with a clever approach by avoiding the issues produced by threads and locks. In the Actor model, all objects are modeled as identical, functional entities that respond only to data received. In addition, the Actor model also has well-defined ways of gracefully

managing errors and failures, maintaining a degree of stability that isolates problems and avoids cascading failures with major downtime. Lastly, one of the most important features of the Actor model is that, actor functions and communicates very much like the way that humans do [4, p.2].

4.1.1 Akka

The Akka framework has adopted the Actor Model paradigm to generate an event-driven, middleware framework for interconnected, scalable, distributed system's development. Akka consists of multiple open-source libraries for developing scalable, robust systems that can utilize processor cores efficiently and span over networks. Akka helps developers to concentrate on business needs instead of writing low-level code to provide consistent behavior, fault tolerance, and high-performance computing features. Akka applies the Actor model to increase the abstraction level that vastly simplifies business logic from low-level lock, threads and non-blocking I/O constructs [5, p.1][6]. The Akka framework sets out the following features:

- Concurrency: Concurrency handling is abstracted, and actor model enables developer to concentrate on the business logic [5, p.1].
- Scalability: Asynchronous messaging allows applications to scale vertically on multicore machines [5, p.1].
- Fault tolerance: Akka leverages Erlang's concepts and techniques to create a "Let It Crash" fault tolerance model using supervisory hierarchies. System can fail quickly and recover as soon as possible from failures [5, p.2].
- Event-driven architecture: Asynchronous messaging nature in Akka allow platform to follow event-driven architectures [5, p.2].
- Transaction support: Akka implements transactors that incorporate actors and STM into transactional actors to allow atomic message flows with automatic retry and rollback [5, p.2].
- Location transparency: Akka handles remote and local process actors equally, offering a single programming model for multi-core and distributed computing needs [5, p.2].
- Scala/Java APIs: Akka allow applications to utilize its Java and Scala APIs [5, p.2]

4.1.2 Event Sourcing and Akka Persistence

Event sourcing is a method to preserve the state of the application by saving the history of events that defines the current application's condition. The primary advantage of the use of event sourcing is a built-in audit mechanism that guarantees the accuracy of transaction and audit data. Representation through events enables user to recreate the state of any entity at any time [7].

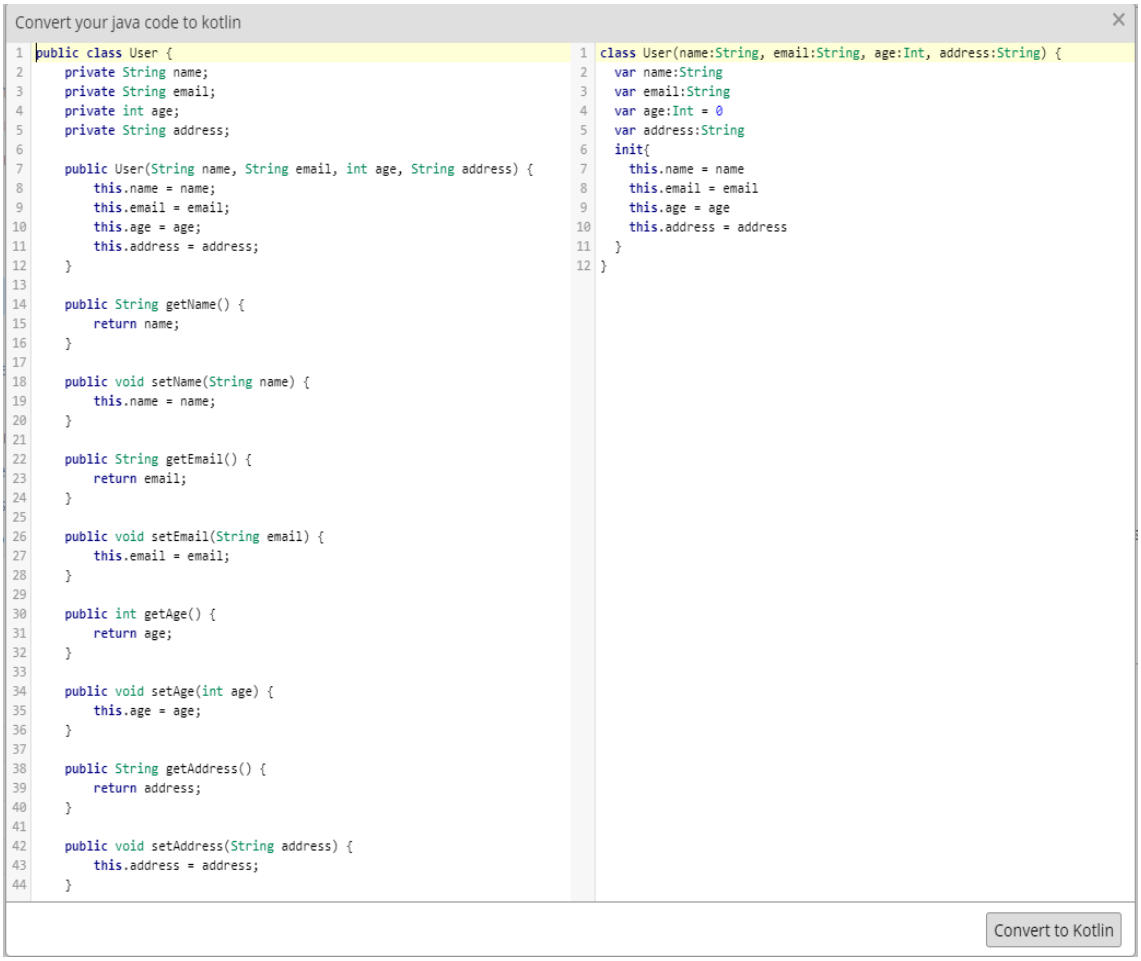
Akka introduces also a sourcing concept through Akka Persistence, which allows stateful actors to preserve their state so that it can be restored when an actor's lifecycle is changed. The main principle behind Akka Persistence is that only events that remain with the actor are stored, not the actual state of the actor. However, actor state snapshot is also supported to significantly reduce recovery times. Events persisted by adding immutably to storage, which enables extreme transaction rates and effective replication. Subsequently, recorded events can be replayed to reconstruct the actor's state or condition [8].

4.2 Kotlin

Kotlin is a new programming language for the Java platform. As an alternative for Java, it was created by JetBrains to reduce the amount of boilerplate code needed and add new features to make the language more descriptive and concise [1, p.3]. Kotlin is chosen to be the main language of the application due to some nice features:

- Kotlin is statically typed and support type inference, allowing it to preserve accuracy and performance while maintaining the source code succinct [7, p.5].
- Kotlin supports both object-oriented and function programming types, allowing higher-level abstractions by first-class functions and testing simplification. In addition, multi-threaded creation is safer and easier by promoting immutable values [7, p.6-7].
- It fits well for server-side applications and completely supported by common existing Java frameworks, including the Spring ecosystem in the IoT platform [7, p.8].
- Kotlin is free and open source. Additionally, it is received a lot of support and development effort from open-source community, major IDEs and build systems [7, p.7].

- Kotlin can interoperate perfectly with Java code [7, p.8], specifically with the Akka used in the platform.



Listing 1. Comparison example between Java and equivalent Kotlin code. Copied from [8].

4.3 Apache Kafka

Apache Kafka is chosen as the main messaging system and potentially evolve into a distributed streaming network for data pipelines and streaming applications on the GFarming IoT network. Apache Kafka was designed to solve the data pipeline issue at LinkedIn and was released as an open source project at GitHub at the end of 2010 [11]. Kafka is chosen as the key messaging system for our follow-up characteristics:

First, Kafka can interact with multiple producers seamlessly, whether they use a wide range of topics or the same subject. In addition to multiple producers, Kafka is designed to allow multiple consumers to read any single stream of messages without disrupting

each other. As shown in figure below, the offset is regulated by the consumer, enabling the user to consume records in any order they like [12, p.10].

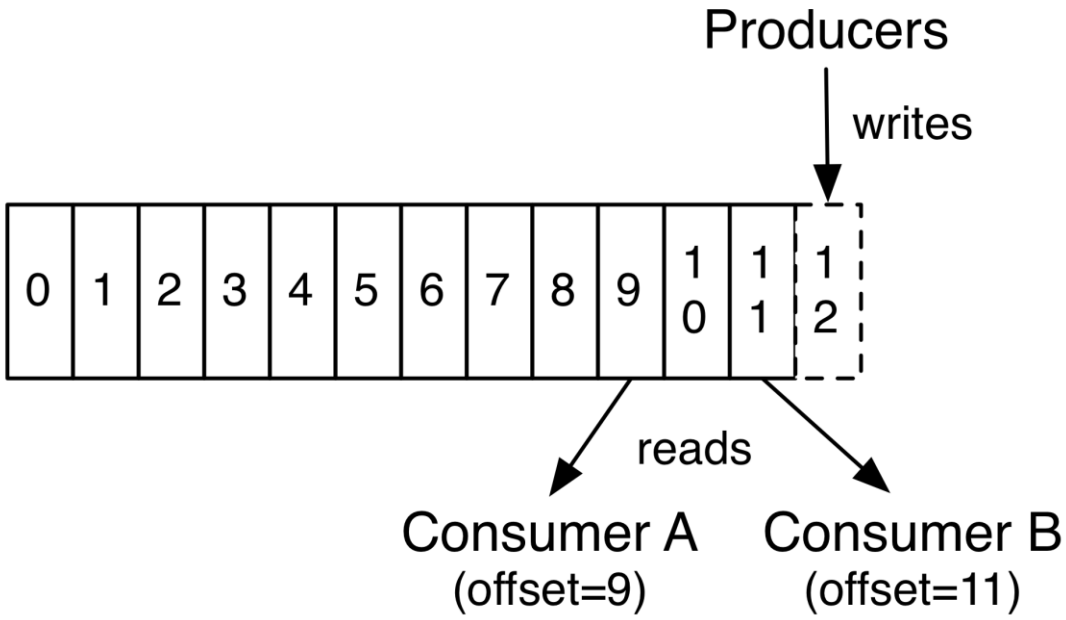


Figure 2. Offset controlled per consumer in Kafka. Copied from [11].

Second, not only can Kafka accommodate many customers, but customers do not always need to work in real time by long-term retention of messages. Messages are written to the disk and will be stored under configurable retention rules. Such choices may be chosen on a per-topic basis, allowing different types of messages to have varying levels of retention depending on the needs of the consumer. Durable retention implies that if the consumer falls behind, either because of slow processing or because of a network issue, there is no chance of losing data. It also ensures that maintenance can be carried out on customers, taking applications offline for a limited period, without any worry over data being buried or lost on producers. Consumers may be stopped, and messages will be left in Kafka. This helps them to restart and catch up processing messages where they paused without data loss [12, p.10].

Third, Kafka's versatile elasticity makes it easy to manage any amount of data. Developers can begin with a single broker and then expand to larger cluster consisting of multiple nodes as data increases. A multi-broker Kafka cluster may manage a broken individual's

failure and continue servicing clients. In addition, clusters that withstand more simultaneous failures can be optimized with higher replication factors [12, p.10-11].

Another important aspect of Kafka is reliability guarantee. First, Kafka makes sure the order of the messages in a partition. If the message A was written before the message B from the same producer in the same partition, the message B offset will be higher than the message A and that the reader will encounter the message A before the message B. Second, generated message is "committed" to the partition when it is written on all of its in-sync replicas. In particular, producers may choose to receive acknowledgment of the sent messages when the message has been received by the leader or sent over the network. Third, messages that have been committed will be reachable when at least one copy remains alive. Finally, Kafka only permits users to read messages that have been committed [12, p.116].

Finally, all these features make Apache Kafka an appealing messaging system that can handle high data rate with outstanding results. Producers, consumers and brokers can all be configured to accommodate very wide streams of messages while guaranteeing the delivery of data to customers in a very small amount of time [12, p.11].

4.4 Apache Cassandra

The explosive number of devices that generate, track and share data across a variety of networks is overwhelming for most data management solutions. Cassandra is included in the project because it is well suited to processing a lot of time-series data coming from gateways and sensors. Apache Cassandra, an open source Apache project that was originally born on Facebook, is a distributed database that stores large amounts of data. Cassandra’s distribution design is based on Amazon’s Dynamo and its data models are inspired by Google’s Bigtable. Many companies have successfully deployed and benefited from Apache Cassandra, including major companies such as Apple, Instagram, Netflix and many others. The larger production environments have Petabytes of data in clusters of more than 75000 nodes [15]. The main features of Apache Cassandra, shown in Figure 4, are extremely valuable to the design and the scalability capability of the IoT platform.

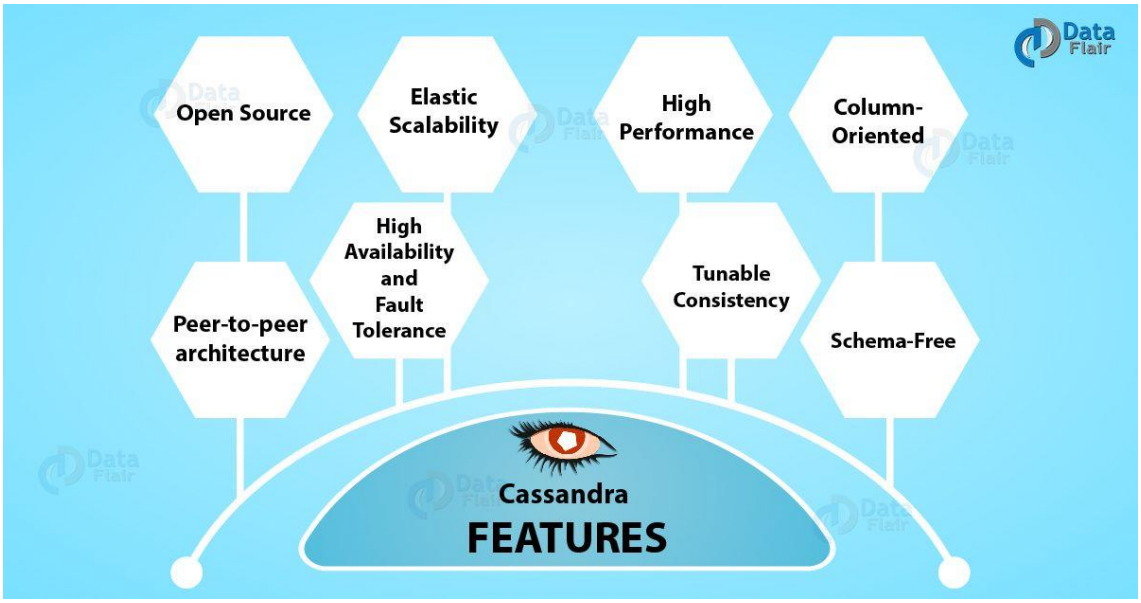


Figure 3. Important Cassandra features. Copied from [13].

Firstly, Apache Cassandra is open source. Therefore, the application is built with lower cost. In addition, there is a big community that can help in maintaining, supporting and developing the platform [13].

Secondly, Cassandra uses peer-to-peer or decentralized architecture, rather than using the traditional master-slave or manual sharded approach. In general, all nodes or servers in the cluster are identical and use gossiping to maintain and sync between nodes. No Cassandra node performs such organizing operations separate from any other node, which means no single point of failure. The decentralized design also enables high availability in Cassandra. In addition, a decentralized cluster is therefore simpler to run and manage, since all nodes are equals [14, p.14-15].

Third, there's elastic scalability in Cassandra. Elastic scalability implies that, depending on the application’s requirement, a cluster can scale up and scale down seamlessly. New nodes can be accepted by the cluster. After that, those can start participating by receiving data and start receiving user’s requests without major interruption or reconfiguration in the exist cluster. Additionally, any of the nodes can be deleted without impacting the functionality of the remainder of the cluster [14, p.16].

In addition to the elastic scalability, Cassandra is also fault tolerant and high available. Any node may encounter all kinds of failures, from failure of hardware to interruption of

the network. However, those failures don't affect the functionality of the cluster as data in Cassandra is distributed automatically to several nodes. Moreover, multiple data centers' replication is also provided to improve local performance and avoid downtime when a failure is faced by one data center. Eventually, failed nodes can be replaced without stopping the whole cluster [14, p.16].

Furthermore, Cassandra requires the client to configure consistency to enable tunable consistency. Cassandra's client can control the number of replicas to block for certain operations by specifying the consistency level with the replication factor. The replication factor determines how much performance will be reduced for more consistency [14, p.19].

In Cassandra, each given row may have one or more columns, and those columns can be different than other rows in the same table. Every row has a unique key that makes its data reachable [14, p.23]. In addition, Cassandra is schema-free, which means that the data structure does not have to be determined specifically before time. That also helps our prototype to grow easily over time. Instead of first modeling data models using costly software and then implementing queries with complicated join statements, Cassandra allow developers to model the queries, and then specify the data around them [14, p.24].

Finally, Cassandra is optimized for high performance by utilizing the multiprocessor machines and run the cluster across many of those machines in multiple data centers. It can scale consistently and seamlessly to handle large amount of data. Cassandra has been proven to operate perfectly with high data traffic. As more nodes are introduced, all of Cassandra's advantageous features are preserved without decreasing performance [14, p.24].

4.5 Spring

Traditionally, Java enterprise application has been often built using the J2EE specification. The J2EE platform provides various communication standards, utilities, and APIs that provide simpler way for multi-tiered, high performance web-based applications development [17]. However, J2EE is still considered to be complex. Spring appeared in

2003 as an attempt to solve problems of the early J2EE specification. Although some consider Java EE and Spring to be competitors, Spring actually complements to the Java EE platform. In details, the Spring programming model does not contain the specification of the Java EE platform; rather, it is designed carefully to work with chosen individual specifications from the J2EE such as Servlet API, Websocket API, Bean Validation or concurrency utilities [18].

Gradually, Java EE plays a crucial role in the application development. During the early days of Java EE and Spring, programs were built to be deployed on an application server. Now, with the aid of Spring Boot, applications can be built in a devops and cloud-friendly manner. Furthermore, Spring begins to develop and grow into a new ecosystem. Besides the Spring Framework, there are other projects, including Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, among others. Each project does contain its own source code version control, bug tracking, and release lifecycle [18].

4.5.1 Spring Framework

The Spring Framework has features that are arranged into 20 modules. These modules are grouped into main 6 categories: Core Container, Data Access / Integration, Web, AOP and Instrumentation, Messaging and Test [19].

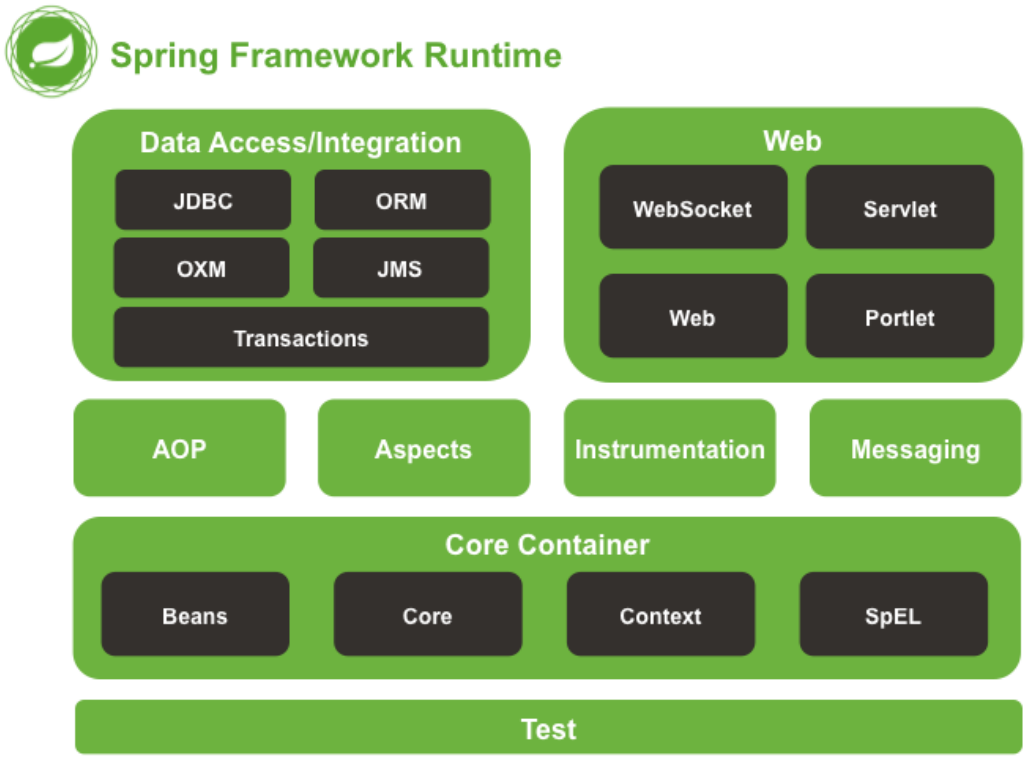


Figure 4. Overview of the Spring Framework. Copied from [19].

Firstly, the Core Container includes the spring-core, spring beans, spring-context, spring-context-support and spring-expression. IOC and Dependency Injection, the foundational features of the framework, are parts of the spring-core and spring-beans modules. Spring-context allows access to application's objects in a framework-style way. Apart from that, spring-context-support includes support for third-party libraries for various purposes. Finally, to fetch and modify object at run time, developers can utilize the compelling expression language retrieved from spring-expression [19].

Secondly, regarding Messaging, Spring framework 4 includes spring-message that can help developing message-based system with message to method wiring annotations. Moreover, application's web layer can be built easily with multiple Web modules, including spring-web, spring-webmvc, spring-websocket, spring-webmvc-portlet. In addition to Messaging and Web layer, AOP and Instrumentation or Data Access/Integrations modules also can be used depends on specific needs on those layers [19].

Lastly, for testing purposes, spring-test is developed with support for unit testing and integration testing [19].

4.5.2 Spring Boot

Spring Boot reduces effort to configure, build and run stand-alone, high-quality Spring-based applications. Most Spring Boot applications require a very small Spring setup [20]. The main goals of Spring Boot are [20]:

- Spring application development can be initialized with quick and straight-forward experience
- Requirements can be expressed freely and achieved quickly even when they start to diverge.
- Non-functional features that are common among applications are added, such as embedded servers, security, metrics, health checks, and externalized configuration.
- Code generation and XML configuration are not required.

4.5.3 Spring Data

The purpose of Spring Data is to provide a familiar and consistent Spring-based data access programming model and to preserve the specific characteristics of the underlying data store. Spring Data makes it easy to apply data access technologies, including relational and non-relational databases, map-reducing frameworks or cloud-based data services. Spring Data for Apache Cassandra is utilized in GFarming IoT application to provide easy configuration and access to Apache Cassandra, the main underling data source [21].

4.5.4 Spring for Apache Kafka

Spring for Apache Kafka is used in the project to support the development process of customers and producers in the application written with the Spring eco system. The Spring for Apache Kafka (spring-kafka) project uses key spring principles to the development of Kafka-based messaging solutions. It offers a template for sending messages as a high-level abstraction and provides support for message driven POJOs with Kafka-Listener annotations as well as a listener container [22].

5 Implementation

5.1 Project setup

Firstly, the project is initialized with Spring Initializr, a generator tool for Spring project. With the help of Spring Initializr, application can be easily generated with various options, including but not limited to supported languages or project's dependencies.

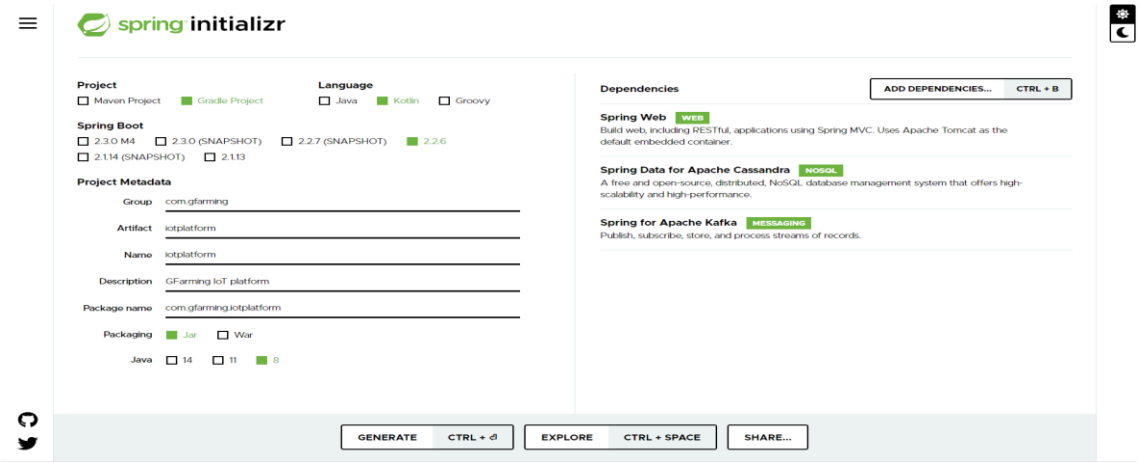


Figure 5. Example of project configuration with Spring Initializr. Copied from [9].

Secondly, Docker is used to arrange application's development environment. Docker Compose is a tool for defining and running multi-container Docker applications [17], in our application, those are Apache Kafka and Apache Cassandra. As illustrated in listing 2 and 3, Cassandra cluster's and Kafka cluster's nodes are configured in the docker-compose.yaml file. Then, with a single command **docker-compose up**, the services are installed with the specified configurations without the need for manual installation.

```yaml
cassandra_node:
  image: cassandra:3.11.6
  container_name: cassandra_node
  volumes:
  - ./cassandra:/var/lib/cassandra
  ports:
  - "9042:9042"
  networks:
    - gfarming
  healthcheck:
    test: ["CMD", "cqlsh", "-u cassandra", "-p cassandra", "-e describe keyspaces"]
    interval: 15s
    timeout: 10s
    retries: 10
```

Listing 2. Cassandra's configuration in docker-compose.yaml.

In listing 3, Zookeeper is also required as it is considered the managing tool for Kafka cluster. All the defined services use the same networks **gfarming**, so they can interact with each other and with the main application.

```yaml
zookeeper:
  image: confluentinc/cp-zookeeper:5.4.0
  hostname: zookeeper
  container_name: zookeeper
  ports:
    - "2181:2181"
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
    ZOOKEEPER_TICK_TIME: 2000
  networks:
    - gfarming
broker:
  image: confluentinc/cp-server:5.4.0
  hostname: broker
  container_name: broker
  depends_on:
    - zookeeper
  ports:
    - "9092:9092"
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092
    KAFKA_METRIC_REPORTERS: io.confluent.metrics.reporter.ConfluentMetricsReporter
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
    KAFKA_CONFLUENT_LICENSE_TOPIC_REPLICATION_FACTOR: 1
    CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: broker:29092
    CONFLUENT_METRICS_REPORTER_ZOOKEEPER_CONNECT: zookeeper:2181
    CONFLUENT_METRICS_REPORTER_TOPIC_REPLICAS: 1
    CONFLUENT_METRICS_ENABLE: 'true'
    CONFLUENT_SUPPORT_CUSTOMER_ID: 'anonymous'
  networks:
    - gfarming
```

Listing 3. Kafka's configuration in docker-compose.yaml.

Finally, the application can be bootstrapped using IntelliJ Idea. The JetBrains IntelliJ IDEA is used as main development editor as it has strong support for Spring application's configuration and various other useful features.

5.2 Domain and Repository Layer

Repositories are classes in common data access layer in Spring application's structure. The usual repository implementation requires a lot of code, often duplicate for common data access behaviors. As stated in chapter 3's Spring Data section overview, the main objective is to provide an intimate and compatible Spring-based model for data accessing pattern in repositories. As showed in listing 4, our data domain objects are created and annotated with Spring metadata annotation. The **@Table** annotation takes full advantage of the object mapping functionality inside the Spring Data for Apache Cassandra support, the classpath scanner will find and pre-process the **Sensor** and **Gateway** data domain objects to extract the necessary metadata such as primary key and Cassandra user type.

```
@Table( value: "gateway")
data class Gateway(
        @PrimaryKey
        val id: UUID,
        val name: String,
        val description: String
)

@Table( value: "sensor")
data class Sensor(
        @PrimaryKeyColumn(ordinal = 1, type = PrimaryKeyType.CLUSTERED)
        val id: UUID,
        @PrimaryKeyColumn(name = "gateway_id", ordinal = 0, type = PrimaryKeyType.PARTITIONED)
        val gatewayId: UUID,
        @CassandraType(type = DataType.Name.UDT, userTypeName = "sensor_type")
        val type: SensorType,
        val name: String,
        val description: String)

@UserDefinedType( value: "sensor_type")
enum class SensorType {
        AIR_TEMPERATURE, SOIL_TEMPERATURE, SOIL_MOISTURE,
}
```

Listing 4. Data definition with spring data annotation.

In the application, **GatewayRepository** and **SensorRepository** are created by extending the Spring Data's Repository abstraction. Repository abstraction allows repository declarations in the data access layer. With the help of repository abstraction, boilerplate

code to implement repositories remarkably diminishes. Listing 5 illustrates the repository interface declaration with a custom finder method, **findByGatewayId**.

.

```
@Repository
interface GatewayRepository: CassandraRepository<Gateway, UUID>

@Repository
interface SensorRepository: MapIdCassandraRepository<Sensor> {
    fun findByGatewayId(gatewayId: UUID): List<Sensor>
}
```

Listing 5. Repositories' definition with **Repository** annotation and abstract classes.

5.3 IoT Actor System

5.3.1 Application actor hierarchy

Actor system is considered as a hierarchical structure, where an actor always belongs to a parent. In addition, an actor is supervised by its parent, which means the parent actor can choose to resume, restart or stop the child actor during failures. As illustrated below, all actors have a common parent, the user guardian, which is defined and created when the ActorSystem is started. Before our application first actor, Akka has already created two actors in the system: the root guardian and the user guardian. Additionally, Akka or other libraries built on top of Akka may generate their own actors under the system guardian [6].

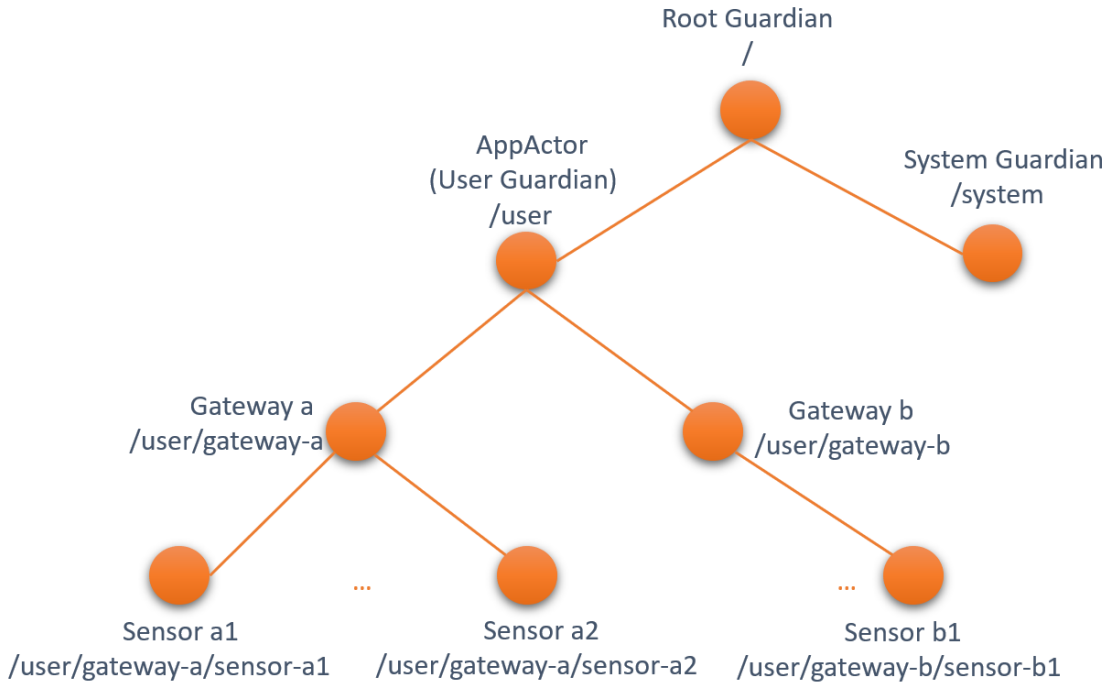


Figure 6. Application's main actor hierarchy.

In order to define an actor, we need to define the type of the messaged handle by the actor. Listing 6 illustrates an example of message type for the application's app actor. The main actor's definition as showed in listing 7, extending the **AbstractBehavior** class, then specifies the logic to handle defined commands and provide corresponding responses.

```
interface Command
class StartAppCmd : Command
data class ToSensorActorCmd(val telemetry: Telemetry,
                            val replyTo: ActorRef<SensorActor.Companion.TelemetryAddedEvt>): Command
data class GetGatewaySensorsStateCmd(val gatewayId: UUID,
                                     val replyTo: ActorRef<GatewaySensorStateResponse>): Command
data class GatewaySensorStateResponse(val telemetries: List<Telemetry>): Command
```

Listing 6. Message type definition for **AppActor**.

```
class AppActor(context: ActorContext<Command>,
               private val appContext: AppContext) : AbstractBehavior<AppActor.Companion.Command>(context) {

    private val gatewayActors: BiMap<UUID, ActorRef<GatewayActor.Companion.Command>> = HashBiMap.create()

    override fun createReceive(): Receive<Command> {
        return newReceiveBuilder()
                .onMessage(StartAppCmd::class.java) { _ -> this.onStartApp() }
                .onMessage(ToSensorActorCmd::class.java, this::onToSensorActorCommand)
                .onMessage(GetGatewaySensorsStateCmd::class.java, this::onGetGatewaySensorsState)
                .build()
    }
```

Listing 7. Main definition for **AppActor**'s behavior.

As mentioned earlier, actor can create, supervise and send message to another actor. Listing 8 describes how app actor can create child gateway actors in **onStartApp** and send messages to appropriate actors in **onToSensorActorCommand**. Specifically, actor can make use of **ActorContext** to spawn new actor and **ActorRef** to send message to other actors.

```
private fun onStartApp(): Behavior<Command> {
    context.log.info("Start App Actor!")
    val gateways = appContext.gatewayRepository.findAll()
    for (gateway in gateways) {
        gatewayActors.computeIfAbsent(gateway.id) { key ->
            context.log.info("Creating gateway actor $key")
            val actorRef = context.spawn(
                    GatewayActor.create(appContext, key, context.self), name: "gateway-$key")
            context.watch(actorRef)
            actorRef ^computeIfAbsent
        }
    }
    return this
}

private fun onToSensorActorCommand(command: ToSensorActorCmd): Behavior<Command> {
    context.log.info("onToSensorActor $command")
    gatewayActors[command.telemetry.gatewayId]?.tell(
            GatewayActor.Companion.TelemetryToSensorCmd(command.telemetry, command.replyTo))
    return this
}
```

Listing 8. Creating actor and sending message in **AppActor**.

5.3.2 Sensor Persistence Actor

Unlike App Actor or Gateway Actor, Sensor Actor is declared by extending the abstract class **EventSourcedBehavior**. The behavior of sensor actor is typed to the type of sensor actor's **Command** defined in the class companion object.

```
class SensorActor (private val sensorId: UUID,
                   private val gatewayId: UUID,
                   private val type: SensorType,
                   private val appContext: AppContext)
    : EventSourcedBehavior<SensorActor.Companion.Command,
      SensorActor.Companion.Event, SensorActor.Companion.State>(

      PersistenceId.ofUniqueId( id: "sensor-$sensorId-gateway-$gatewayId"),
      SupervisorStrategy.restartWithBackoff(
              Duration.ofSeconds( seconds: 10),
              Duration.ofSeconds( seconds: 30),
              randomFactor: 0.2)) {


    override fun emptyState(): State {
        return State(Telemetry(gatewayId, sensorId, type, value: null, timestamp: null))
    }
```

Listing 9. Constructor and **emptyState** implementation in **SensorActor**.

As illustrated in listing 9, **SensorActor** needs to define the required unique **PersistenceId**, which is the combination of **sensorId** and its belonging **gatewayId**. In addition, **emptyState** override method defines the State when the sensor is created, that is telemetry with **null** values. Another important note in the sensor actor implementation is the appearance of **SupervisorStrategy**. Supervision is a mechanism in Akka to handle fault tolerance. With regards to the sensor actor case, if an exception is thrown from the journal, that actor will be restarted with the calculated durations from the **restartWithBackoff**'s arguments.

```
override fun commandHandler(): CommandHandler<Command, Event, State> {
    val b: CommandHandlerBuilder<Command, Event, State> = newCommandHandlerBuilder()
    b.forAnyState()
            .onCommand(AddTelemetryCmd::class.java, this::onAddTelemetry)
            .onCommand(CurrentStateRequest::class.java, this::onCurrentStateRequest)
    return b.build()
}

private fun onAddTelemetry(state: State, cmd: AddTelemetryCmd): Effect<Event, State> {
    LOGGER.info("onAddTelemetry sensor $sensorId gateway $gatewayId")
    return Effect().persist(TelemetryAddedEvt(cmd.telemetry))
            .thenRun {
                cmd.replyTo.tell(TelemetryAddedEvt(cmd.telemetry))
                appContext.messagingTemplate.convertAndSend(
                        destination: "/gateway/$gatewayId/telemetry",
                        Collections.singletonList(cmd.telemetry))
            }
}

private fun onCurrentStateRequest(state: State, cmd: CurrentStateRequest): Effect<Event, State> {
    LOGGER.info("onCurrentStateRequest sensor $sensorId gateway $gatewayId")
    return Effect().reply(cmd.replyTo, CurrentStateResponse(state.telemetry))
}


override fun eventHandler(): EventHandler<State, Event> {
    return newEventHandlerBuilder().forAnyState()
            .onEvent(TelemetryAddedEvt::class.java) {state, event ->
                state.telemetry = event.telemetry
                LOGGER.info("Updated sensor $sensorId gateway $gatewayId state $state")
                state ^onEvent
            }
            .build()
}
```

Listing 10. Command handling and state updating logic inside **SensorActor**.

Method **commandHandler** described in listing 10 defines the mechanism to handle incoming **AddTelemetryCmd** and **CurrentStateRequest** commands. During add telemetry event, the actor will persist corresponding **TelemetryAddedEvt**, then reply to request actor and notify WebSocket subscribers as side effects.

Another important logic in sensor actor, the way state is updated and recovered, is defined in **eventHandler**. The state of the actor is simply the new telemetry added event's value.

5.3.3 Kafka consumer's implementation

In Kafka, before publishing and receiving data through producers and consumers, a topic should be defined. In Spring Kafka, a new topic can be defined simply by adding **NewTopic** **@Bean** to the application context [22].

```
// DEFINING TOPIC
@Bean
fun telemetryTopic(): NewTopic {
    return TopicBuilder.name(TELEMETRY_KAFKA_TOPIC)
            .partitions( partitionCount: 3)
            .replicas( replicaCount: 1)
            .compact()
            .build()
}

/**
```

Listing 11. Defining Kafka Telemetry Topic as Spring bean.

As shown in listing 11, **TopicBuilder** is used to construct the needed **NewTopic** more conveniently. The current configuration defines the partitions to 3 and replicas to 1 respectively. Those are suited for now to the prototype's scope; however, those should be modified in later stage to reach production level. Replication is at the core of Kafka's architecture and is the way Kafka enable availability and durability during specific node's failure. Data in Kafka is split into several topics, each topic is then divided into multiple partitions, and each partition can be configured with multiple replicas. Replicas of various topics and partitions gradually ends up being stored on Kafka brokers [12, p.97].

As illustrated in listing below, Spring Kafka consist of convenient annotation **@KafkaListener** to implement consumer's behavior method. In the context of **listen** method, the received data will be logged and send to **actorService** for further processing.

```
@KafkaListener(topics = [TELEMETRY_KAFKA_TOPIC])
fun listen(telemetry: Telemetry,
           ack: Acknowledgment,
           @Header(KafkaHeaders.RECEIVED_PARTITION_ID) partitionId: Int) {
    LOGGER.info("on telemetry from partition $partitionId")
    actorService.onTelemetrySensorAdded(telemetry, ack)
}
```

Listing 12. Defining Kafka consumer for telemetry topic.

5.3.4 Adding telemetry to sensor actor

When interaction to actor system is needed from Kafka consumer, which located outside of the actor system, the ask version in Akka **AskPattern** return a **CompletionStage<Response>** that is either finished with a successful response or failed with a TimeoutException if there was no response within the specified timeout [24]. Figure 7 describes the message flow from Kafka components to the actor system. In details, message request is delivered between multiple actors, finishes in sensor actor for persisting. After that, response is sent back to request actor in Kafka Consumer for acknowledgement to Kafka brokers. Importantly, the use of actors allows asynchronous processing and request thread can accept other tasks while waiting for response.

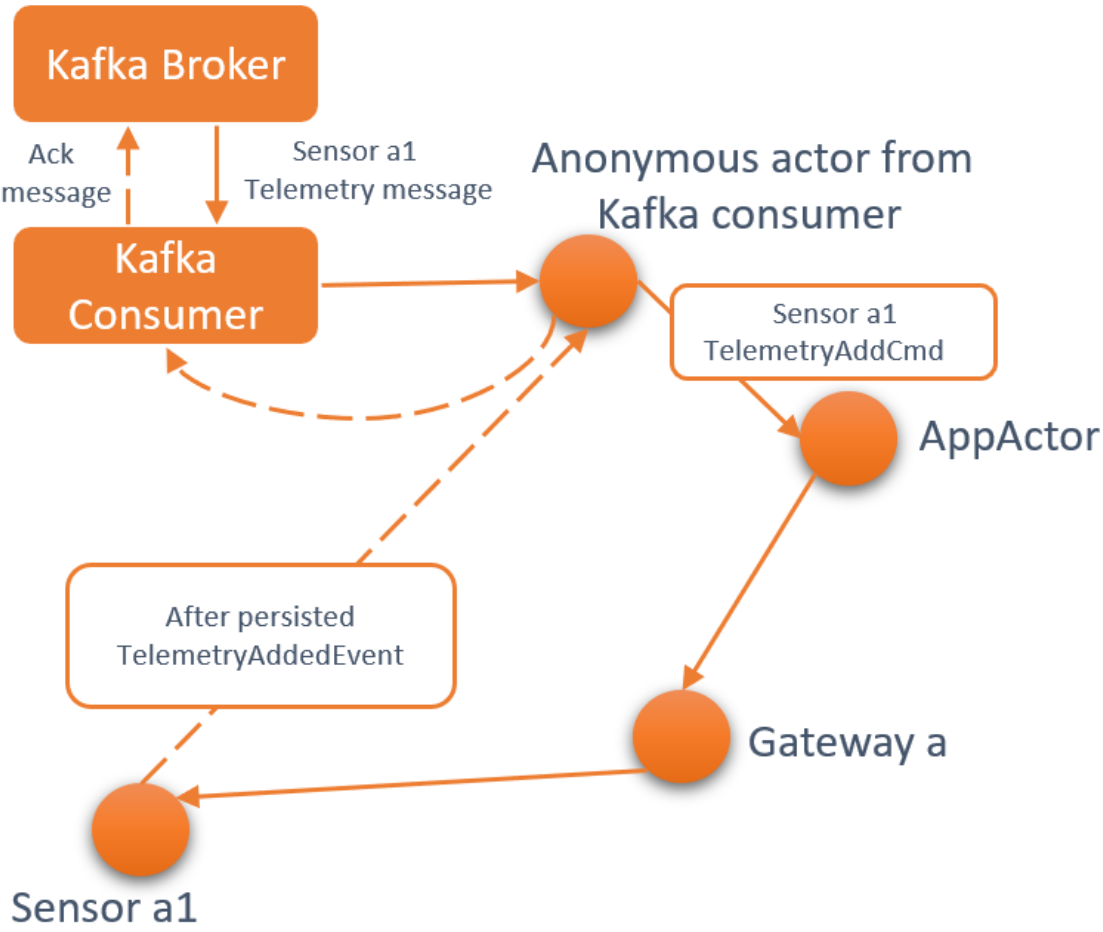


Figure 7. Request-response flow between actors when adding gateway's telemetry from Kafka.

5.3.5 Querying gateway sensors state

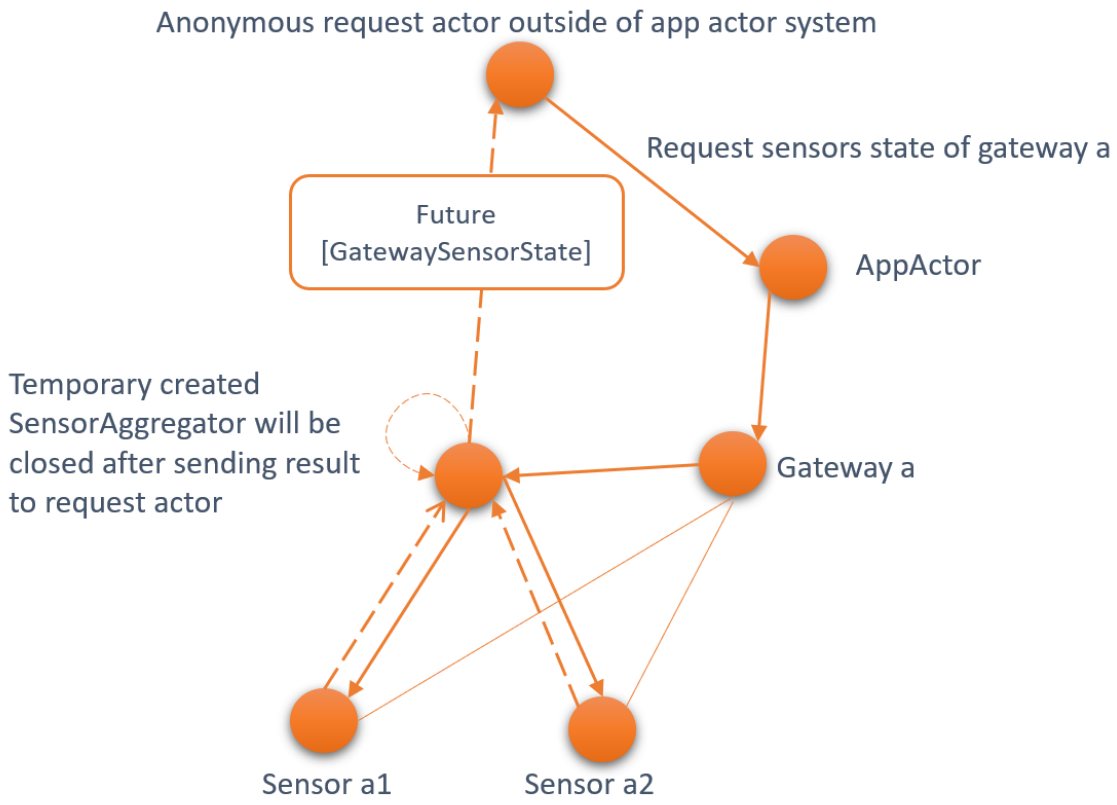


Figure 8. Request-response flow between actors to get gateway sensors state.

As described in figure 7, similar to the logic in adding telemetry, Akka **AskPattern** with anonymous request actor will send a request with gatewayId to the app actor system. After that, corresponding gateway will end up handle this request by spinning off an actor dedicated for sending and collecting state from the gateway's child actors. After collecting successfully states from all child actors, the sensor aggregator actor will send response with the **GatewaySensorState** back to the anonymous request actor. By creating a dedicated **SensorAggregatorActor**, the gateway actor doesn't need to handle all the requests and aggregation logic. Meanwhile, it can perform other processing job as telemetry keep coming continuously. In addition, the sensor aggregator will handle any possible time out or failure coming from gateway's child actors.

5.4 WebSocket Integration

WebSocket protocol establishes an important new feature for web applications: full-duplex, two-way interaction between client and server. Spring Framework contains spring-WebSocket module with inclusive WebSocket support. It follows the Java WebSocket API standard and implements also extra features [25]. WebSocket is a great fit to IoT application where the client and server need to exchange telemetry events at high frequency and with low latency.

To enable web socket support in the application, the **WebsocketConfig** configuration class extends the abstract **WebSocketMessageBrokerConfigurer** is created. The built-in simple message broker is used for handling clients' subscription requests, record them in memory, and broadcasts data to connected clients with identical destinations [25].

```
@Configuration
@EnableWebSocketMessageBroker
class WebsocketConfig: WebSocketMessageBrokerConfigurer {

    override fun registerStompEndpoints(registry: StompEndpointRegistry) {
        registry.addEndpoint( ...paths: "/gfarming")
                .setAllowedOrigins("*")
                .withSockJS()
    }

    override fun configureMessageBroker(registry: MessageBrokerRegistry) {
        registry.enableSimpleBroker( ...destinationPrefixes: "/gateway")
    }

}
```

Listing 13. WebSocket endpoint and broker configuration.

The communication flow during gateway telemetry subscription is described in figure below.

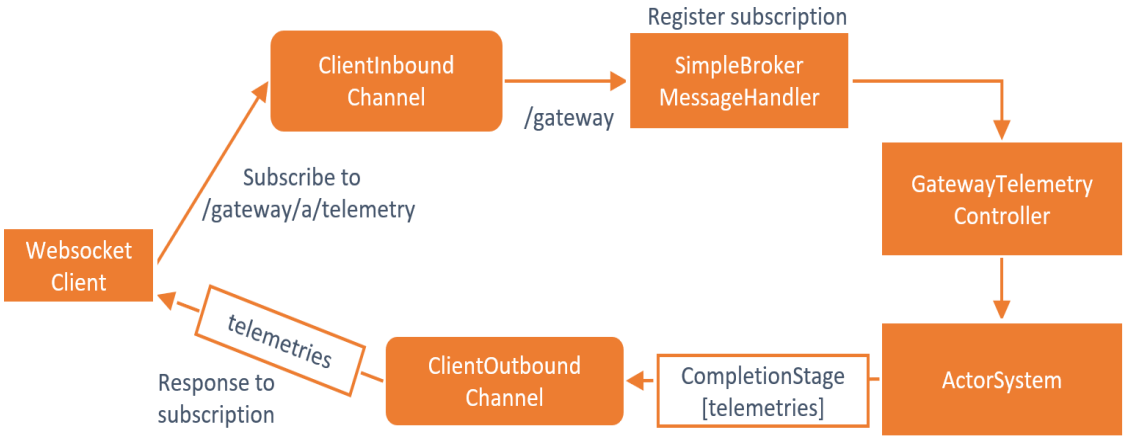


Figure 9. Gateway Telemetry Subscription's communication flow.

As defined in figure 9, subscription messages from WebSocket client will be received on the **clientInboundChannel** and forwarded to the broker to be handle on **SimpleBrokerMessageHandler**. The subscription will be registered in Spring default subscription registry, after that, handler defined with **@SubscribeMapping** in **GatewayTelemetryController** will be triggered. The return value of **@SubscribeMapping** method by default is sent directly to the client through **clientOutboundChannel**, in response to the subscription. Essentially, this behavior suits the feature of populating UI WebSocket client initial data upon first telemetry topic subscription. Subsequent delta publishing is done in **SensorActor** after telemetry persisting through **SimpMessagingTemplate** as shown in listing 10. Spring supports handling response message asynchronously with type **ListenableFuture**, **CompletableFuture** or **CompletionStage** [25] as in our case.

5.5 Unit Testing

Testing the controller endpoints can be done with help of **WebMvcTest** and **MockMvc**. Full auto-configuration is turned off in @WebMvcTest annotation test. Instead, only configuration related to the MVC test is applied including a **MockMvc** instance [26]. The instance of MockMvc can be used to perform testing endpoint request and expecting the test response behaviors, as show in listing 14.

```
@Test
fun getGatewaySensorsStateTest() {
    val gatewayId = UUID.randomUUID().toString()
    val telemetries: List<Telemetry> = arrayListOf(TestHelper.generateTelemetry())
    `when`(actorService.getSensorsStateOfGateway(gatewayId))
            .thenReturn(CompletableFuture.completedFuture(telemetries))

    val mvcResult = mockMvc.perform(get( urlTemplate: "/gateway/$gatewayId/telemetry"))
            .andExpect(status().isOk)
            .andExpect(request().asyncStarted())
            .andExpect(request().asyncResult(telemetries))
            .andReturn()

    mockMvc.perform(asyncDispatch(mvcResult))
            .andExpect(status().isOk)
            .andExpect(content().string(
                    TestHelper.objectMapper.writeValueAsString(telemetries)))
}
```

Listing 14. **getGatewaySensorsStateTest** with WebMvcTest and MockMvc.

In addition to rest controller testing behavior, the main functional requirements in actor system are tested. The request and response of an anonymous actor for adding telemetry and querying gateway's sensor state are verified with the help of Akka's **ActorTestKit** as shown in listing 12. **ActorTestKit**'s main benefit is testing actor's behavior in an asynchronous and realistic way [27].

```
// create and start app actor
val messagingTemplate: SimpMessagingTemplate = Mockito.mock(SimpMessagingTemplate::class.java)
val appContext = AppContext(gatewayRepository, sensorRepository, messagingTemplate)
val appActor = testKit.spawn(AppActor.create(appContext), name: "appActor")
appActor.tell(AppActor.Companion.StartAppCmd())

// test add telemetry to actor system
val addTelemetryActor: TestProbe<SensorActor.Companion.TelemetryAddedEvt> = testKit.createTestProbe()
appActor.tell(AppActor.Companion.ToSensorActorCmd(telemetry, addTelemetryActor.ref))
addTelemetryActor.expectMessageClass(SensorActor.Companion.TelemetryAddedEvt::class.java)

// test get gatewaySensorsState
val requestActor: TestProbe<AppActor.Companion.GatewaySensorStateResponse> = testKit.createTestProbe()
appActor.tell(AppActor.Companion.GetGatewaySensorsStateCmd(gatewayId, requestActor.ref))
val response: AppActor.Companion.GatewaySensorStateResponse = requestActor.receiveMessage()
assertEquals( expected: 1, response.telemetries.size)
assertEquals(telemetry, response.telemetries[0])
```

Listing 15. Actor system's test.

6 Testing and Conclusion

6.1 Testing

Because a concrete UI implementation is absent, application rest endpoints' behaviors are tested through Swagger UI tool. As showed in listing 16, the Swagger UI exposes different endpoints related to gateway, sensor and telemetry. In addition, interacting with those endpoints can also be done directly in the UI, an example illustrated in figure 11.

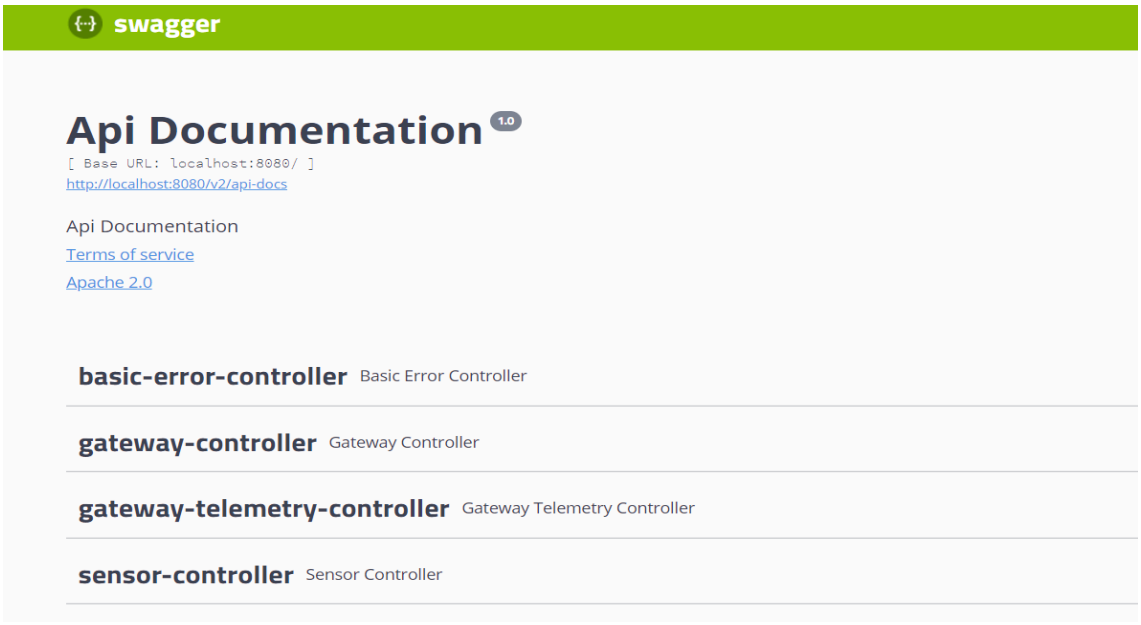


Figure 10. Application's Swagger UI.

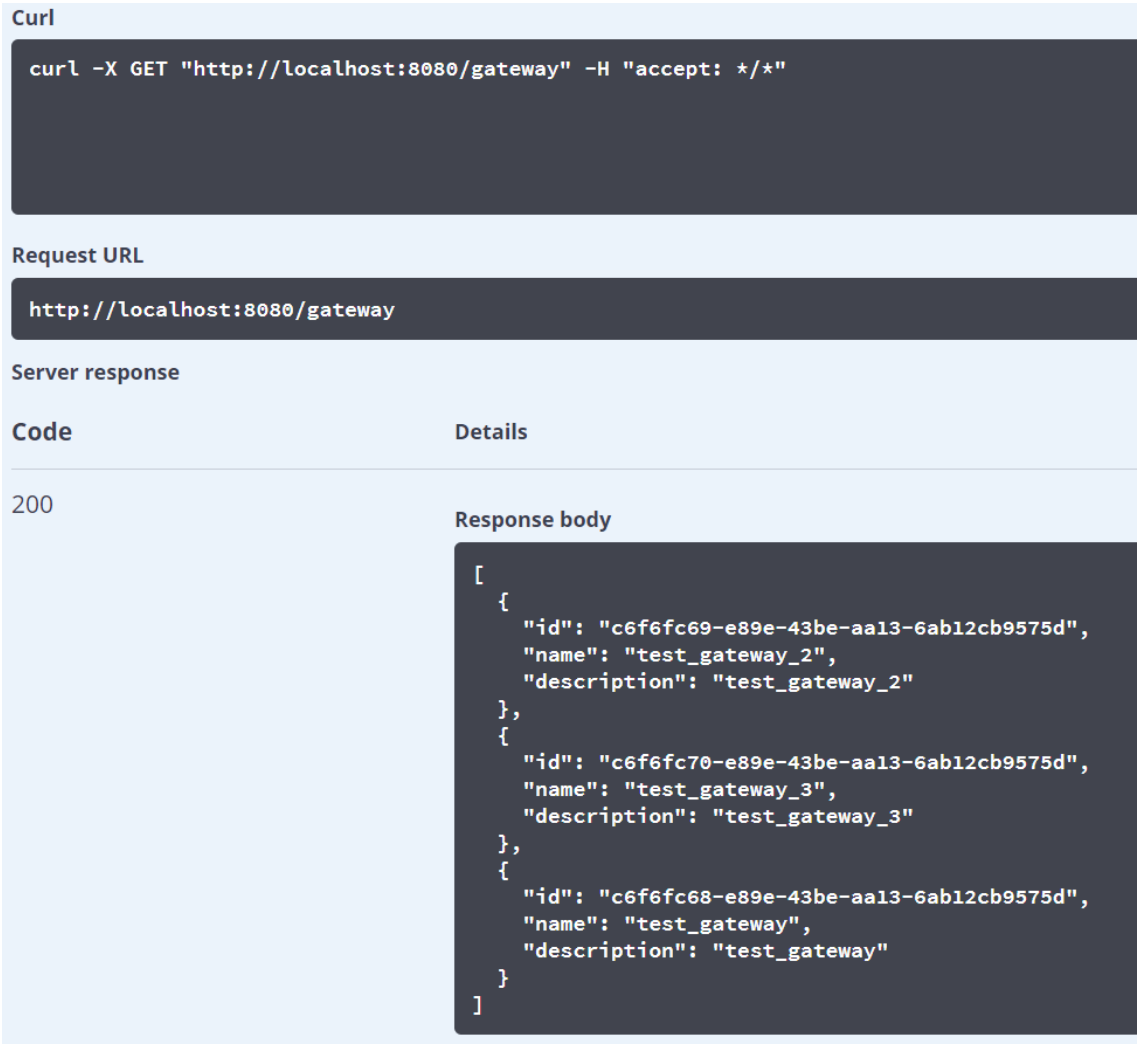


Figure 11. Fetching gateways example in Swagger UI.

WebSocket endpoints, however, are tested through a simple JavaScript client. The stimulation producers are configured to run on scheduler for the demo in the main Spring application. Directing to appropriate gateway telemetry URL allows data to be updated to the client with correct behavior as in figure 12.

Figure 12. Updates coming from telemetry subscription in Angular client.

6.2 Conclusion

To sum up, despite facing many challenges during the implementation, such as steep learning curve or some documentation's unclarities, the IoT Platform prototype for GFarming is successfully developed using many modern tools and an approach to reactive system architecture. Moreover, the application can be further developed and scale as need. The source code in Kotlin is also concise, straightforward and can be developed with ease by other developers. The platform can now be integrated with the company's developed sensors and gateways for various purposes. Furthermore, the prototype also shows the suitability of Actor model and various modern distributed systems in development of an IoT solutions.

However, due to limited scope of the thesis and the prototype's requirements, there are many areas and features in the application that can be improved. A CI/CD pipeline, exception handling, data validation, adding Kafka and Cassandra nodes or a rule engine actor system are some concrete examples that enable the application to be in a production ready stage. Additionally, the application's load can be spread to multiple nodes or servers by integrating Akka Cluster Sharding with the consistent hashing algorithm. Finally, a client-side component written in modern JavaScript framework can be added to showcase multiple appealing features of the platform to the end users.

References

1. The Fundamental of IoT Architecture [online]. https://www.losant.com/blog/the-fundamental-iot-architecture. Accessed on 19 April 2020.

2. Smart Agriculture Monitoring solutions to optimize farming productivity [online]. URL: https://easternpeak.com/blog/smart-agriculture-monitoring-solutions-to-optimize-farming-productivity/. Accessed on 19 April 2020.

3. The Reactive Manifesto [online]. URL: https://www.reactivemanifesto.org/. Accessed on 15 March 2020.

4. Hugh McKee. Designing Reactive Systems, the Role of Actors in Distributed Architecture. O'Reilly; 2016.

5. Munish K. Gupta. Akka Essentials – A practical, step-by-step guide to learn and build Akka's actor-based, distributed, concurrent, and scalable Java applications. Packt; 2012.

6. Introduction to Akka – Akka Documentation [online]. URL: https://doc.akka.io/docs/akka/current/typed/guide/introduction.html. Accessed on 15 March 2020.

7. Event Sourcing [online]. URL: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj591559(v=pandp.10)?redirectedfrom=MSDN. Accessed on 22 April 2020.

8. Event Sourcing – Akka Documentation [online]. URL: https://doc.akka.io/docs/akka/current/typed/persistence.html. Accessed on 22 April 2020.

9. Dmitry Jemerov, Svetlana Isakova. Kotlin in Action. Manning; 2017.

10. Try Kotlin [online]. URL: https://try.kotlinlang.org/. Accessed on 18 April 2020.

11. Apache Kafka Introduction [online]. URL: https://kafka.apache.org/intro. Accessed on 15 March 2020.

12. Neha Narkhede, Gwen Shapira, and Todd Palino. Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale. O'Reilly; 2017

13. 5 important Cassandra features that you must know – DataFlair. [online]. URL: https://data-flair.training/blogs/cassandra-features/. Accessed on 19 April 2020.

14. Eben Hewitt. Cassandra: The Definite Guide. O'Reilly; 2011.

15. Apache Cassandra [online]. URL: http://cassandra.apache.org/. Accessed on 15 March 2020.

16. Spring Initializr [online]. URL: https://start.spring.io/. Accessed on 18 April 2020.

17. Java 2 Platform, Enterprise Edition (J2EE) Overview [online]. URL: https://www.oracle.com/java/technologies/appmodel.html. Accessed on 23 April 2020.

18. Spring Framework overview [online]. URL: https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html. Accessed on 23 April 2020.

19. Spring Framework Reference Documentation [online]. URL: https://docs.spring.io/spring/docs/4.3.9.RELEASE/spring-framework-reference/htmlsingle/. Accessed on 1 March 2020.

20. Spring Boot Reference Documentation [online]. URL: https://docs.spring.io/spring-boot/docs/2.2.6.RELEASE/reference/html/. Accessed on 23 April 2020.

21. Spring Data [online]. URL: https://spring.io/projects/spring-data. Accessed on 23 April 2020.

22. Spring for Apache Kafka [online]. URL: https://spring.io/projects/spring-kafka. Accessed on 23 April 2020.

23. Overview of Docker Compose [online]. URL: https://docs.docker.com/compose/. Accessed on 21 April 2020.

24. Interaction Patterns – Akka Documentation [online]. URL: https://doc.akka.io/docs/akka/current/typed/interaction-patterns.html. Accessed on 23 April 2020.

25. WebSocket Support [online]. URL: https://docs.spring.io/spring-framework/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/websocket.html. Accessed on 19 April 2020.

26. WebMvcTest (Spring Boot Docs 2.2.6.RELEASE API) [online]. URL: https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/autoconfigure/web/servlet/WebMvcTest.html. Accessed on 23 April 2020.

27. Asynchronous Testing – Akka Documentation [online]. URL: https://doc.akka.io/docs/akka/current/typed/testing-async.html. Accessed on 23 April 2020.